Prof. Dr. Arnd Poetzsch-Heffter

Kurs 01798

Software-Architektur

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Liebe Fernstudentin, lieber Fernstudent,

dieses Dokument stellt eine Leseprobe für den Kurs 1798 "Software-Architektur" dar. Die Probe besteht aus

- dem Vorwort, das die Ziele des Kurses, zentrale Fragestellungen und den Aufbau der Kurseinheiten vorstellt,
- dem Inhaltsverzeichnis,
- dem Literaturverzeichnis,
- Studierhinweisen zur Kurseinheit 1,
- den ersten beiden Unterkapiteln von Kapitel 1 "Software Architecture: An Introduction",
- Überblicken über die weiteren Kapitel des Kurses,
- einer Einsendearbeit.

Wir hoffen, Ihnen damit einen realistischen Einblick in den Kurs bieten zu können.

Viele Grüße aus Hagen

Daniela Keller, LG Programmiersystene, daniela.keller@fernuni-hagen.de

Vorwort

Liebe Fernstudentin, lieber Fernstudent,

wir begrüßen Sie zum Kurs "Software-Architektur", der Teil des Moduls "Vertiefung Software Engineering und Programmiersprachen" ist, und wünschen Ihnen viel Spaß mit dem Kurs.

Ziel des Kurses

Software-Systeme sind immer größer und komplexer geworden, wodurch verstärkt Fragestellungen in den Vordergrund getreten sind, die sich mit globalen Systemeigenschaften und der Struktur, Organisation und dem Aufbau von Systemen beschäftigen. Von besonderem Interesse sind dabei die Kommunikation zwischen Systemteilen und die Konstruktion von Systemen aus Komponenten. Demgegenüber spielen bei der Entwicklung derartiger Systeme algorithmische Aspekte und die Auswahl geeigneter Datenstrukturen zunächst eine untergeordnete Rolle.

Die Architektur eines Software-Systems beschreibt,

- aus welchen Komponenten das System besteht,
- wie diese Komponenten zusammengesetzt sind,
- welche Funktionalität sie besitzen
- und ggf. wie sie auf Rechner an verschiedenen Orten zu verteilen sind.

Die Architektur legt die globalen Kontrollstrukturen fest und regelt Kommunikation, Synchronisation und Datenzugriff zwischen den Komponenten.
Sie sollte Auskunft über die Leistungsfähigkeit des Systems geben.

Wenn man von der Architektur eines Software-Systems spricht, konzentriert man sich also auf das Zusammenwirken der Komponenten und abstrahiert von deren Realisierungsdetails.

Software-Architektur ist ein Teilgebiet der Software-Technik. Zentrale Fragestellungen sind:

- Wie beschreibt und bewertet man Software-Architekturen?
- Wie spezifiziert man Komponenten, wie deren Zusammenwirken, wie deren Zusammensetzen?
- Gibt es typische Muster, die man immer wieder in Software-Architekturen vorfindet bzw. für deren Entwurf gebrauchen kann?

- Wie klassifiziert man solche Muster?
- Gibt es Richtlinien, nach denen man Architekturmuster für den Entwurf von Software-Systemen systematisch kombinieren kann?
- Welche Auswirkungen haben Änderungen der Systemanforderungen auf eine Architektur?
- Ist eine Architekur bzgl. ihrer Leistungsfähigkeit skalierbar?

Ziel des Kurses ist es, Ihnen Antworten auf diese und weitere Fragen zu geben und Ihnen damit eine erste Einführung in das Fach Software-Architektur zu bieten.

Studienmaterial und Aufbau des Kurses

Der Kurs besteht aus vier Kurseinheiten mit den folgenden Titeln:

- KE1: Architektur von Software-Systemen: Eine Einführung
- KE2: Sichten, Aspekte und Muster in Software-Architekturen
- KE3: Generische Architekturen und Komponentensysteme
- KE4: Beschreibung und Entwurf von Architekturen

Zu jeder Kurseinheit gibt es einen **Kurstext in Englisch** mit der Angabe der **Lernziele** sowie **Einsendeaufgaben**, die Sie selbständig bearbeiten sollen und zur Korrektur einsenden können. Die Lösung der Einsendeaufgaben ist wesentlicher Bestandteil des Kurses.
Darber hinaus gehört zum Kurs das **Studium von Fachaufsätzen und Fallstudien**, die wir Ihnen in der virtuellen Universitt bereitstellen.

Inhaltsverzeichnis

1 Software Architecture: An Introduction **3**
1.1 Subject Matter and Goals 3
1.1.1 Software Systems 4
1.1.2 Structures of Software Systems 6
1.1.3 Goals and Problems of Software Architecture 7
1.2 Architecture of Software Systems 11
1.2.1 The Architecture of a Software System 11
1.2.2 Generic Architectures and Patterns 15

2 Software Systems and Architecture **17**
2.1 A Closer Look at Software Systems 17
2.2 Examples of Software Systems and Architectures 17

3 Architectural View and Aspects **18**

4 Program Frameworks **19**

5 Architectures for Component Software **20**

6 Description Techniques for Architectures **21**

7 Designing Software Architectures **23**

Contents

1 Software Architecture: An Introduction **3**
1.1 Subject Matter and Goals 3
1.1.1 Software Systems 4
1.1.2 Structures of Software Systems 6
1.1.3 Goals and Problems of Software Architecture 8
1.2 Architecture of Software Systems 11
1.2.1 The Architecture of a Software System 11
1.2.2 Generic Architectures and Patterns 14
1.3 Relation to other Disciplines 15
1.4 Overview and Further Reading 17

2 Software Systems and Architecture **19**
2.1 A Closer Look at Software Systems 19
2.1.1 Software Systems and their Contexts 19
2.1.2 Characteristics of Software Systems 21
2.1.3 What is Software? What is a Software System? 23
2.2 Examples of Software Systems and Architectures 25
2.2.1 GCC: The GNU Compiler Collection 26
2.2.2 A Three Tier Architecture 26

3 Architectural View and Aspects **33**
3.1 Architectural Structures and Views 33
3.1.1 An Introduction to Views 34
3.1.2 The Dynamic View of Software Systems 36
3.1.3 The Static View of Software Systems 38
3.1.4 The Relationship between Dynamic and Static View . . . 39
3.1.5 An Example Based Discussion of Views 42
3.2 Architectural Aspects 49
3.2.1 Availability 50

4 Architectural Patterns **53**
4.1 Introduction 54
4.2 Layered Organization 56

4.3 Repositories . . . 57
4.4 Compound Object Pattern . . . 59
4.5 Pipes and Filters . . . 60
4.6 Process-Control Pattern . . . 61

5 Program Frameworks 67
5.1 Introduction to Program Frameworks . . . 67
5.2 A Program Framework for User Interfaces . . . 70
5.2.1 Architectural Properties of Graphical User Interfaces . . 70
5.2.2 The Abstract Window Toolkit of Java . . . 73

6 Architectures for Component Software 79
6.1 Component Frameworks: An Introduction . . . 80
6.2 Microsoft's Component Object Model . . . 82
6.2.1 Component Model . . . 83
6.2.2 Component Infrastructure . . . 85
6.2.3 Further Features of the COM Framework . . . 88
6.3 Enterprise JavaBeans . . . 89
6.3.1 Annotations . . . 91
6.3.2 Component Model . . . 92
6.3.3 Entities . . . 97
6.3.4 Component Infrastructure . . . 100

7 Description Techniques for Architectures 109
7.1 Unified Modeling Language: An Overview . . . 110
7.1.1 UML Diagrams . . . 110
7.1.2 UML Class Diagrams . . . 113
7.1.3 UML Interaction Diagrams . . . 115
7.1.4 Unified Modeling Language as ADL . . . 117
7.2 Architectural Frameworks . . . 118
7.2.1 An Introduction to Architectural Frameworks . . . 118
7.2.2 A Tiny Architectural Framework . . . 119
7.3 Connectors in Architectural Descriptions . . . 125

8 Designing Software Architectures 127
8.1 Software Design and Architecture . . . 127
8.1.1 Methods and Techniques for Design . . . 128
8.1.2 The Role of Architectural Structures for Design . . . 130
8.2 Architectures and Evaluation . . . 131
8.2.1 General Design Rules for Architectures . . . 131
8.2.2 Architectures and their Relation to Requirements . . . 132

Literaturverzeichnis

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BW97] M. Büchi and W. Weck. A plea for grey-box components. Technical Report No. 122, Turku Center for Computer Science, Turku, Finland, 1997.

[DP00] S. Denninger and I. Peters. *Enterprise JavaBeans*. Addison-Wesley, 2000.

[FS97] Martin Fowler and Kendall Scott. *UML Destilled*. Addison-Wesley, 1997.

[Gea97] David M. Geary. *Graphic Java: Mastering the AWT*. Prentice Hall, 1997.

[GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *European Conference on Object-Oriented Programming, ECOOP '93*, volume 707 of *Lecture Notes in Computer Science*, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMW00] David Garlan, Robert Monroe, and David Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Kru95] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[LAK+95] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[MH99] Richard Monson-Haefel. *Enterprise Java Beans*. O'Reilly & Associates, 1999.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the Fourth ACM Symposium on the Foundation of Software Engineering*. ACM Press, October 1996.

[Nag90] Manfred Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Springer Compass. Springer-Verlag, 1990.

[ONE] Open net environment (ONE) software architecture. Available at `www.sun.com/software/ whitepapers/index.xml#4`

[SD00] Johannes Siedersleben and Ernst Denert. Wie baut man Informationssysteme? *Informatik Spektrum*, 23(6):247–257, August 2000.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[Ull88] J. D. Ullman. *Database and Knowledge-base Systems, Volume 1: Classical Database Systems*. Computer Science Press, 1988.

[UML99] OMG Unified Modeling Language (version 1.3). Available at `www.rational.com/uml/resources/documentation/index.jsp`, 1999.

[EJB06] Sun Microsystems. JSR 220: Enterprise JavaBeans^{TM}, Version 3.0, May 2006.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification, Third Edition*. Sun Microsystems, 2005. ISBN: 0-321-24678-0.

[JMS] Sun Microsystems. The Java EE 5 Tutorial, Basic JMS API Concepts. Available at http://java.sun.com/javaee/5/docs/tutorial/doc/JMS3.html.

[SGS06] I. Schulz-Gerlach and H.W. Six. Software Engineering II, 2006. Lecture at the University of Hagen.

[SBS06] R.P. Sriganesh, G. Brose, and M. Silverman. *Mastering Enterprise JavaBeans^{TM} 3.0*. Wiley Publishing, Inc., 2006. ISBN: 0-471-78541-5.

Studierhinweise zur Kurseinheit 1

Diese erste Kurseinheit umfasst

- die Kapitel 1 und 2 des nachfolgenden Kurstextes;
- die Skizze einer Architekturbeschreibung: "Call Center Customer Care System (a case study)". Begleitmaterial zum Kurs 1798, 5 Seiten (erhältlich in der Virtuellen Universität, nicht in der Leseprobe enthalten).
- den Technischen Bericht:
  Paul C. Clements, Linda M. Northrop: "Software Architecture: An Executive Overview". Technical Report CMU/SEI-96-TR-003; Begleitmaterial zum Kurs 1798, 44 Seiten. Zugreifbar unter:

  http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.003.html

  Die Abschnitte 4 und 5 des Berichts bereiten dabei bereits auf die 2. Kurseinheit vor.

Die Kurseinheit bietet Ihnen eine Einführung in das Gebiet der Software-Architektur und versucht den Begriff des Software-Systems zu präzisieren.

Lernziele:

- Was ist Software-Architektur? Worum geht es in diesem Fachgebiet? Wofür ist es von Bedeutung?
- Was ist die Beziehung zwischen Software-Architektur und Software-Engineering?
- Was ist ein Software-System?
- Kriterien für die Klassifikation und den Vergleich von Architekturen.
- Beispiele unterschiedlicher Software-Systeme

Lesen Sie zunächst die Kapitel des Kurstextes; studieren Sie dann die Architekturskizze und versuchen Sie, die erlernten Begriffe darauf anzuwenden.
Beachten Sie: Die in den Abbildungen als ASOS bezeichnete Komponente wird im Text NOSS genannt.
Arbeiten Sie im Anschluss den Aufsatz durch.
Schließlich sollten Sie die Einsendeaufgaben lösen.

Kapitel 1

Software Architecture: An Introduction

There is no standard, universally-accepted answer to the question “What is Software Architecture”. Different people still use the term in different ways. However, most people in the field will agree that software architecture is a subdiscipline of software engineering with at least the following goals:

1. Software architecture should provide knowledge about existing, well-designed architectures, frameworks, and architectural patterns.

2. It should develop techniques, methods, models, description languages, and tools to improve understanding, analysis, design, construction, evaluation, and evolution of software systems.

This chapter should help you to develop a coherent conception of what software architecture is about. It approaches this question from three sides: What is the subject matter of software architecture and what are its problems and goals? What is the architecture of a software system? What is the relation to adjacent or enclosing disciplines? Finally, Section ”overview” provides a short overview of the other chapters of the course and recommends further reading.

1.1 Subject Matter and Goals

Scientific or engineering disciplines can be characterized by their subject matter, their goals, and problems. The artifacts studied within software architecture are software systems. The following subsections will sketch several examples of software systems, give an idea of different structuring techniques, and summarize the central goals and problems of the discipline of software architecture.

1.1.1 Software Systems

A *software system* is a technical system or subsystem in which software plays a dominant role. In Chapter 2, we will describe in some detail what we regard as software and how software systems can be classified. Here, we start the discussion of software systems by considering some examples. They provide a background for the remainder of the introduction. In particular, it should become clear that software systems may have very different characteristics. Consider the following examples:

1. A GNU Chess installation on a given computer: It is an interactive system to play chess with the computer. It is a simple, single user application with a graphical user interface running in one process that has no persistent state between executions.

2. The Pascal compiler installed on a given computer: It is an input/output system, taking Pascal programs as input and producing assembler or machine code. It is a batch-oriented, single user system that performs a complex transformation running in one process.

3. The operating system installed on a given computer: It is an interactive platform system for program execution, for file and network access, and for handling input (from keyboard, mouse, microphone,...) and output (to terminals, loudspeakers,...). It is a nonterminating system. It can be shut down and has persistent state between executions (typically information stored in files on harddisks). It defines what a process is; its kernel part belongs to every process; other parts run in separate processes. It provides a rich programming interface and is built to provide a platform for other systems.

4. The World Wide Web, a distributed client-server system for making documents accessible over the Internet. It is an open, extensible platform system with thousands of servers and millions of clients. It is distributed all over the world and provides no central or coordinated activity (no one knows the state of the system).

5. The reservation system of the Deutsche Bahn AG: It is a system for reserving seats in trains. It is a closed, dedicated, and distributed system that provides a number of coordinated services for a company based on user interaction.

6. The D2 telecommunication network, a system for keeping track of the location of mobile phones, for connecting the caller to the callee, for maintaining phone connections, and for accounting. It is a distributed system that cannot be shut down for maintenance. It has to be fault-tolerant and highly available.

7. The automatic pilot of an Airbus A 320, a system for navigating the aircraft towards a destination. It is a nonterminating, embedded realtime system consisting of sensors, computer hardware, and control devices being part of the whole aircraft system. It has to be a highly dependable system.

8. The software control system of an airbag in a car, an embedded realtime system.

All of the above examples are *installed* or *running* software systems, i.e. software installed or executed on one or several physically identifiable platforms (e.g. for the first example, the platform is the operating system installed on the given computer; for the third example, it is the hardware system of the computer). Together with their platform(s), they constitute *operational* systems that provide some tasks/services. From the above examples, it should be clear that software systems are built on top of or are subsystems of other systems.

Software systems can be very complex. It is hardly possible to grasp their overall characteristics and structures by simply looking at the coding of the software. In contrast to concrete engineering results like buildings or bridges, one cannot walk around a software system to get a first idea of it; one cannot use physical dimensions (like length, height, width; weight) to talk about it. However, in most respects software engineering is like other engineering disciplines: Many people are involved in the design, construction, testing, maintenance, and use of software systems. They need means to communicate about those issues of a system that are relevant for them in a precise way. For example:

- Designers have to be able to analyze and compare different designs.
- Programmers have to understand the interfaces and behavior of system components.
- The overall structure of program modules is needed for integration testing, maintenance, and bug elimination.
- The conceptual characteristics and structures are often helpful to explain the use of the system.

One cannot expect that all the issues sketched above can appropriately be captured by one model or one description technique. In practice, a number of models and description techniques is applied to express different structures of a system on different levels of abstraction. Two different ways to abstract a software system are explained in the following subsection.

1.1.2 Structures of Software Systems

Different stakeholders are interested in different views and aspects of a software system. In this subsection, we illustrate this by sketching the conceptual and implementation view of a compiling system that translates Ada and C programs into assembler code[1]. The conceptual view is helpful to understand what a system is supposed to do and how its subtasks are structured. The implementation view explains the structure of the program modules implementing the system.

Figure 1.1 provides the conceptual view of the compiling system. In the figure, boxes denote functional units, arrows indicate data flow between the units, and annotations explain the tasks of the units and the kind of data.

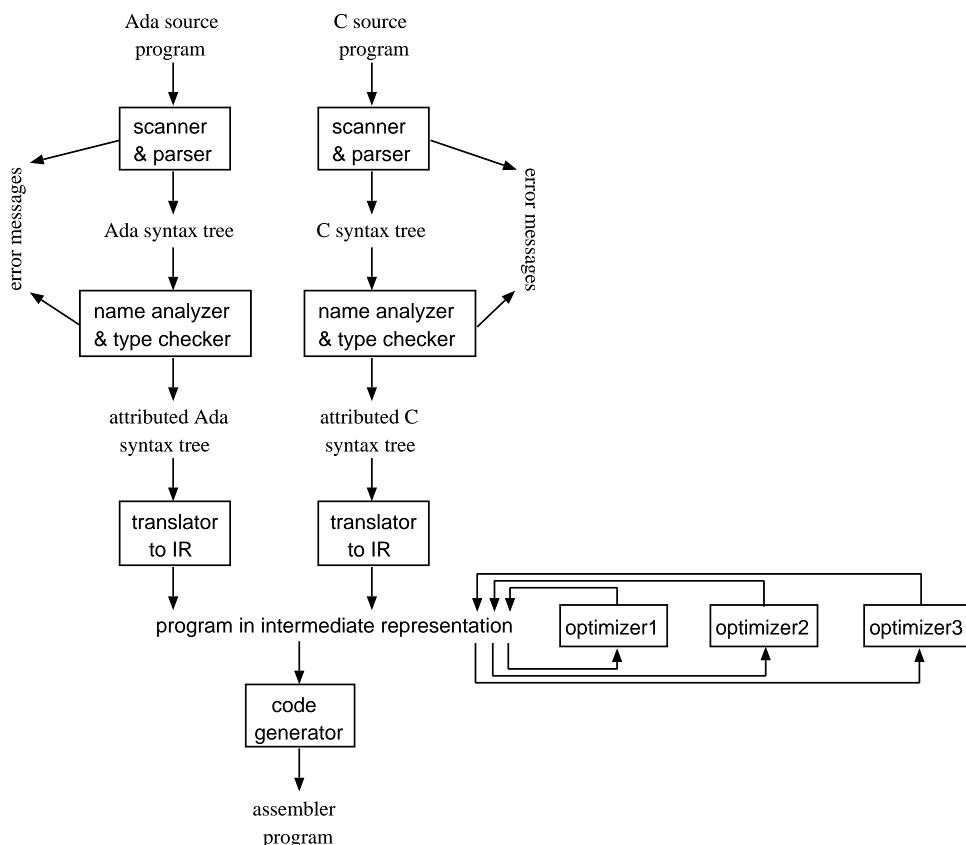

Abbildung 1.1: Conceptual view of a compiling system for Ada and C

From a conceptual point of view, the functional units performing semantic analysis of Ada programs are different from those analysing C programs (semantic analysis includes scanning, parsing, name analysis, and type

[1]We use only two source languages to keep the diagrams simple; an extension to other languages is straightforward.

checking). After semantic analysis, the attributed syntax tree is translated into a language-independent intermediate representation. The three optimization units work on this representation. Finally, the code generator produces the assembler code. An advantage of such an architecture is that the optimizers and the code generator can be used for several source languages. A disadvantage is that specific language features are difficult to exploit for optimization and code generation. The conceptual view provides a basis for high-level design. It describes the relationship between important data structures and functional units and the general data flow of the system.

The implementation view, as given in Figure 1.2, captures the structure of the program modules used. In Figure 1.2, boxes denote program modules whereby boxes with broken lines indicate a hierarchical module structure. Bold arrows specify that one module uses the other, and dotted arrows indicate that a module is automatically generated from a specification.

The implementation view shows the important program modules of the compiling system and their static dependencies. There is a generated Ada scanner and a hand-written C scanner. They are used by a table-driven parser with separate tables for Ada and C syntax. The parser constructs the syntax tree of the input program. An attribute evaluator controls the attribute computation for Ada and C programs using the language-specific attribution procedures and produces the intermediate representation. The optimizers work on the intermediate representation and two of them compute and exploit data flow information. The code generator accesses the intermediate representation and the data flow information.

The implementation view is helpful for all tasks related to the realization of the system. In particular, it provides an overall orientation for accessing the program modules and for understanding their interfaces. Notice that the implementation view does not explain the data flow of the system. Both views focus on different structures of the compiling system. To understand the architecture of the system, both views have to be taken into account. As we will show during this course, it is not always evident what the appropriate and relevant structures of a software system are. In particular, the two structures above are only used here to illustrate what we mean by the structures of a software system. It is not claimed that they capture all architectural properties of the compiling system.

1.1.3 Goals and Problems of Software Architecture

A central task of software architecture is to collect, describe, and refine the knowledge about well-designed architectures, frameworks, and architectural patterns. The goal is to improve the support for routine design, i.e. design in which large portions of prior solutions can be reused. In other engineering disciplines and in building architecture, the study of existing solutions

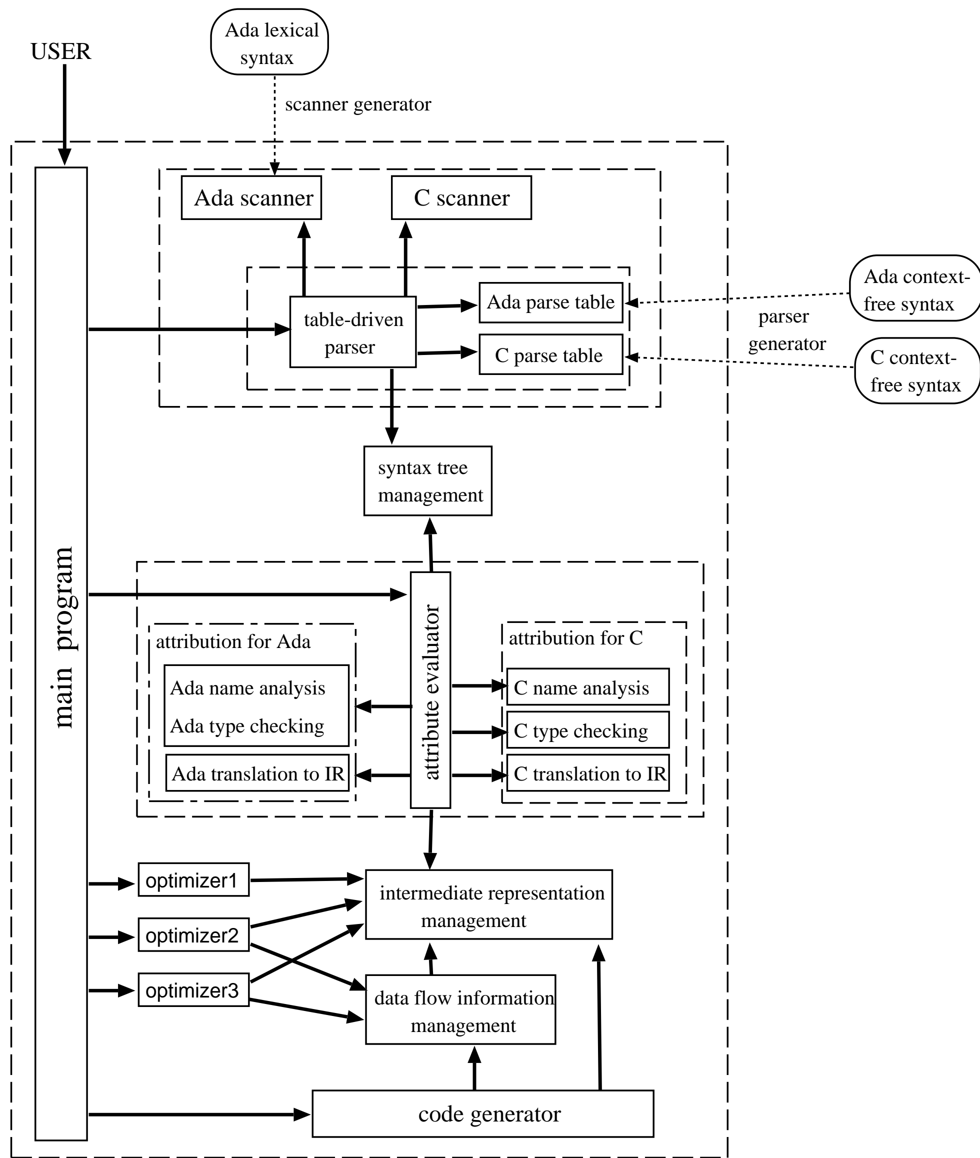

Abbildung 1.2: Implementation view of a compiling system for Ada and C

plays an important role. For computer science this is true for algorithms and data structures, but not yet for higher-level system structures and architectural aspects. Based on this knowledge, software architecture should develop techniques, methods, models, description languages, and tools to improve understanding, analysis, design, construction, evaluation, and evolution of software systems. In order to approach these goals, quite a number of hard and soft problems have to be solved. In the following, we try to sketch some of them.

Description Techniques. Techniques and languages to describe the architectures of software systems are still in their infancy. As we tried to illustrate in the compiler example above, a software system has several relevant structures. These structures have to be captured in a consistent way by architectural description techniques. The problem is to define the semantics of the notations for the different structures in such a way that the relationship between the structures is well-defined and that they complement each other to avoid redundancy. What is even more challenging are the very different characteristics of software systems:

- Systems that are simple applications running in one process (e.g. the chess program).
- Nonterminating systems that are platforms for applications (like operating systems).
- Systems that are distributed over large networks with more or less tightly coupled processes (e.g. the reservation system mentioned above or the Web).

Description techniques have to cover wide classes of architectures. Otherwise they will not be accepted in practice, because no one is willing to learn a particular description technique for each architecture.

Foundations. Software architecture is about the structures of software systems. Structures are explained by decomposing the system into components and by defining the interface behavior of components and their relationship in an abstract way. Whereas we have fairly good techniques to specify concrete systems for which many design and implementation decisions are taken, it is not so clear how to define component interfaces and interactions on a more abstract level. E.g. we can specify the procedural interface of a component and the use of such an interface. However, for most cases, the conceptual structure of an architecture is not the appropriate abstraction level to commit oneself to a special form of communication between components. This decision should be postponed to maintain flexibility for later design stages. For

example, the conceptual structure of the compiler system of Fig. 1.1 expresses the communication between the units in the same way. In the implementation, it is realized differently (cf. Fig. 1.2).

A still needed theoretical foundation of software architecture has to solve the following problems:

1. How can interfaces of computational components and the connections between them be modeled and specified on an abstract level? It should be avoided to commit oneself too early to specific design decisions. What are the needed abstractions? What is their relation to the concrete communication and composition techniques?

2. How can descriptions of different system structures be embedded into a common formal framework?

3. How can architectural descriptions be combined with formal methods so that system properties can be formally verified?

4. How can efficiency or security analysis be based on architectural descriptions?

Reference Architectures, Frameworks, Patterns. The study of successful software architectures should provide reusable solutions to specific design problems. This can be done at different levels. Reference architectures or domain-specific architectures should help in the design of whole systems or large subsystems within specific application domains. Typical examples would be

- the reference architecture for compilers based on the phases scanning, parsing, semantical analysis, optimization, and code generation;

- domain specific architectures for the realization of Web Services (cf. [ONE]) or for business software (cf. [SD00]).

By frameworks we mean generic architectures together with extensible implementations of the kernel functionality for these architectures. Typical examples are frameworks for graphical user interfaces (cf. [Gea97]) or component software (cf. [Szy97]). On a smaller scale, the study of architectures should develop knowledge about reusable patterns that capture the relation between generic program components or objects. Architectural patterns can guide the design phase of a software system and provide a vocabulary to talk about architectural structures (cf. [GHJV93, BMR+96]).

Tools and Visualization. Architectural models and patterns may be helpful for designers even if they are not written down. Architectural descriptions written down in a semi-formal or formal notation are important documents for the development and maintenance of software systems. Nevertheless, the real challenge is to integrate architectural descriptions into the mechanized software development and construction process. With an advanced tool support, they can be used to automatically generate communication code and module interfaces. They provide the basis for system-wide consistency checking, e.g. type checking of module interfaces written in different programming languages. They can even be used to automate system distribution, installation, and startup. Of course, this scenario presupposes sufficiently detailed architectural descriptions and a powerful tool support. In addition to that, architectural descriptions can provide a basis for visualization and testing of complex software systems.

The list of goals and problems of software architecture is not meant to be complete. It is intended to give an idea of the relevant topics in the field.

1.2 Architecture of Software Systems

The last section gave a short sketch of what the *discipline "software architecture"* is about. In this section, we strive to answer the question what the *architecture of a given software system* is. Starting from this answer, we generalize the notion to classes of systems and system patterns.

1.2.1 The Architecture of a Software System

As a first step towards understanding what the architecture of a software system is, let us compare the architecture of a software system to the architecture of a skyscraper, i.e. an artifact of software architecture to an artifact of building architecture. In most cases, an installed software system consists of megabytes of data and binary code, sitting on some harddisks or other storage media of one or several computers. If it is executing, part of its code is loaded into main memory. The skyscraper consists of millions of bricks or tons of concrete, sitting on solid ground. If it is functioning, it needs power supply for air conditioning, elevators, etc. Architecture is certainly not about the basic concrete elements of the systems like bytes or bricks.

The architecture of the skyscraper is about its form and statics, about the structure of its surface, about the internal structuring of the building into floors and rooms, about the organization of staircases and elevators for managing the thousands of people entering and leaving the building every day, about the connection to the outside, about the structure of the air condition,

heating, and power supply, etc. Notice that several of these structures are hidden once the building is finished. For all of these aspects, the architects of the skyscraper will produce a documentation. It is evident that these documents are based on different techniques. However, the documents have to be closely related to describe the architecture of one skyscraper.

The architecture of a complex software system is based on a number of different structures as well. The following list names the relevant components and their relationship for each structure (cf. [BCK98], sect. 2.5):

- *Conceptual structure*: Its components are abstractions of the system's functional requirements. These abstractions are related by the exchanges-data-with relation.
- *Data flow*: Expresses the sends-data-to relation on system components on different levels of abstraction.
- *Control flow*: Expresses the becomes-active-after relation on system components on different levels of abstraction.
- *Hierarchical structure*: Expresses a hierarchical structuring of the systems in subsystems by an is-subsystem-of relation. At the bottom of the hierarchy are program modules.
- *Uses/call structure*: The components are program modules, classes, and procedures. It expresses the import relation on modules, uses relation on classes, and the call relation on procedures.
- *Process structure*: The components are processes and threads. Typical relations are synchronizes-with, cannot-run-without, starts, stops, suspends, or similar relations.
- *Physical structure*: Describes the distribution of code and data to the underlying platforms.

The list is not meant to be complete. For particular systems, additional structures may be useful. E.g. in a telecommunication network that cannot be shut down for maintenance, a special system management structure could be made explicit. On the other hand, different structures can be described together. What we called the conceptual view in the above compiler example combined conceptual and data flow aspects. The implementation view combined aspects of the hierarchical structure and uses structure.

From the analogy to building architecture it should be clear that a short definition of software architecture capturing all its aspects cannot be given. With this in mind, we present three textbook definitions and shortly discuss them. In Section 2.1 of [BCK98], p. 23, the following definition is given:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The definition is given together with an explanation parts of which we like to cite as well:

> Ëxternally visibleproperties refers to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction. Let us look at some of the implications of this definition in more detail.
>
> First, *architecture defines components*. The architecture embodies information about how the components interact with each other. This means that architecture specifically *omits* content information about components that does not pertain to their interaction. Thus, an architecture is foremost an *abstraction* of a system that suppresses details of components that do not affect how they use, are used by, relate to, or interact with other components. [...]
>
> Second, the definition makes clear that *systems can comprise more than one structure*, and that no one structure holds the irrefutable claim to being *the* architecture. [...] By intention, the definition does not specify what architectural components and relationships are. Is a software component an object? A process? A library? A database? A commercial product? It can be any of these things and more.
>
> Third, the definition implies that every software system has an architecture, because every system can be shown to be composed of components and relations among them. [...]
>
> Fourth, the behavior of each component is part of the architecture, insofar as that behavior can be observed or discerned from the point of view of another component. This behavior is what allows components to interact with each other, which is clearly part of the architecture. Hence, most of the box-and-line drawings that are passed off as architectures are in fact not architectures at all. They are simply box-and-line drawings. [...]

The introduction of [SG96], p. 3, gives the following explanation:

> The *architecture of a software system* defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and

> layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database-accessing protocols, asynchronous event multicast, and piped streams.
>
> In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions. At the architectural level, relevant system-level issues typically include properties such as capacity, throughput, consistency, and component compatibility.

In [BMR+96], a book concentrating on patterns, the following definition can be found on page 384:

> A *software architecture* is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity.

If we talk about the architecture of a software system, we refer to the different implicit and explicit structures underlying the system. Structures are expressed as components and their connections or relations. I.e. the architecture captures system properties that are beyond algorithms and data structures. This is the focus of the first definition and is a central part of most explanations of software architecture.

The second definition mentions another important and often neglected point. Architecture is about relevant system-level or system-wide issues and how they are realized within the system. In addition to the properties named above, these include aspects like security, system availability, and fault tolerance. Thus, architecture is not only about the structuring of systems, but captures as well the relationship between the overall system behavior and the structured realization.

The third definition contains a remark about the nature of architectures: They are artifacts, i.e. something explicit, the result of an activity. Notice that this idea is contradictory to the third implication above saying that every software system has an architecture, possibly an implicit one. However, it is an important issue that architectures can be artifacts in their own right. They can be designed and studied without going down to concrete realizations in the form of software systems. This is the topic of the next subsection.

1.2.2 Generic Architectures and Patterns

So far, we looked at the architecture of single, installed software systems. In these cases, an architecture has a concrete counterpart, namely a physical installation on computers. In practice, this concrete notion is only relevant for very large systems for which there is a small number of installations (e.g. the World Wide Web, or the D2 telecommunication network). In general, *abstractions of this concrete notion* are much more important.

The simplest step is to abstract from the *installation* of a software system. E.g. we can talk about the architecture of the GNU Chess system without referring to the installation on a specific computer. This makes sense, because the installation of GNU Chess has no influence on architectural issues. Similarly, we can describe the architecture of a specific version of the Linux operating system without refering to an installation. However, in this case the situation is slightly different. It is an interesting architectural property that the kernel of Linux can be configured to special needs during installation. Thus, architectural properties are lost in the installation process. We say a software system is *implemented* if it is completely programmed, but not yet installed. As suggested by the examples, we talk as well about the architecture of implemented software systems. The next abstraction steps are fairly straightforward: We can abstract from different configurations of software systems (e.g. whether they are configured for different platforms) and from different versions. In many cases, neither the configuration nor the specific version of a system has an impact on its architecture.

More generic are *product-line architectures* that describe the architectural structures of a family of software products within a company or organization. The idea is that similar products can be developed and maintained more cost-effectively if they share a common design and parts of the implementation (cf. [HNS00], p. 7, and [BCK98], Chapter 15). Further abstraction steps underly so-called *domain-specific architectures* and *reference architectures*. A domain-specific architecture captures the known architectural abstractions specific to a given problem domain. Reference architectures are often considered even more abstract, capturing some high-level design decisions and standards that a group of people or organizations have agreed on. Some authors consider domain-specific and reference architectures as synonyms.

Whereas the above architectural notions refer to families or abstractions of whole software systems, *architectural patterns* describe useful organizational schemata to structure software systems. According to the definition given in [BMR+96], an architectural pattern *provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.* We use architectural patterns and architectural styles as synonyms.

We introduced the above notions like product-line or reference architec-

ture as abstractions from the architecture of an installed software system. We did so because it is often simpler to go from the concrete to the more abstract. However, this should not imply a direction how architectures are developed. It may be the case that a product-line architecture is designed as a generalization of the architecture of a specific product. But it is as well possible and can be advisable that the architecture of a specific product is obtained by instantiating and refining a pre-existing product-line architecture.

Kapitel 2

Software Systems and Architecture

This chapter has two topics:

1. It states more precisely what a software system is and how the term "software" is used in the following.

2. It describes two examples in order to illustrate the introduced concepts and as background material for the remaining chapters.

2.1 A Closer Look at Software Systems

Software architecture is about architectural aspects of single software systems, of classes of software systems, of incomplete and extensible software systems, and about patterns occurring in software systems. Here, we focus on architectures of single software systems. In the first subsection, we distinguish between the software system of interest, its platform and its system environment. Then, we review important characteristics of software systems. Finally, we discuss terminology related issues; in particular, we explain what we mean by software.

2.2 Examples of Software Systems and Architectures

In this section, we first describe the GNU compiler collection to illustrate the concepts and notions introduced so far and to provide an example for a software system family. Then, we explain a typical three tier architecture. As a third example, we refer to the description of the Call Center Customer Care System (cf. Studierhinweise).

Kapitel 3

Architectural Views and Aspects

In Subsection ”architecture-of-a-sotfware-system”, we sketched different ways to structure software systems and explained that an architecture captures as well the relationship between the overall system behavior and the structured realization. In the following we aim at developing a systematic approach for these issues. We distinguish between two principles for the presentation and documentation of an architecture. The first principle explains a system by structuring its behavior or its software. Typical questions are:

What are the states of a system?
What are its components?

The second principle explains a system by focusing on different system-wide properties like performance, availability, scalability, or security. A typical question is:

How does the interaction of system components guarantee an acceptable response time for the services provided by the system?

The two principles are complementary in the following sense. On the one hand, system-wide properties might be better investigated based on the decomposition of the system into components (e.g. availability of the overall system can be derived from the availability of its components and their relations). On the other hand, the system-wide properties define the criteria for a good decomposition (e.g. the decomposition of a system usually affects the performance or the scalability of a system).

In the following section, we will consider structuring mechanisms according to the first principle. In Section ”architectural-aspects”, we will sketch the investigation of system-wide properties which are often called architectural aspects.

Kapitel 4

Program Frameworks

Many software systems have similar subtasks, use common communication mechanisms, share components, or are based on the same architectural pattern. E.g. many software systems have a graphical user interface; many systems communicate via remote procedure call; applications might share components like a spell checker or a search engine; many systems are organized according to the three-tier architectural pattern.

If systems share properties, there is a chance to reuse architectural and design knowledge, implementation techniques, implementation parts, and communication infrastructure. An important method to support reuse is the construction of frameworks that capture the shared properties. In this chapter, we concentrate on program frameworks in which the shared properties are expressed by implemented program parts. After a general discussion of program frameworks and their relation to patterns and architectures, we consider the Abstract Window Toolkit of Java as a program framework example. In Chapter 5, we provide an introduction to component frameworks.

Kapitel 5

Architectures for Component Software

This chapter introduces component frameworks. Component frameworks support the composition of software systems from concrete software components. Such software components are well-defined entities of their own that can be provided by other development groups or companies. In particular, component technology is important to create a market for prefabricated software. Accordingly, component frameworks have three main goals:

- Reuse of software components.
- Providing the basis for a market of software components.
- Role separation.

Role separation means that the process of developing, deploying, administering, and using a software system can be clearly separated into different roles. Component frameworks enable to separate the responsibility of developing and managing a software system into five roles: Component provider, framework provider, application provider (constructs an application from components), application deployer (installs an application on a platform), and system administrator. Role separation is important for division of labor, specialization, and better competition in the software market.

The chapter focuses on the technical aspects of component software. It starts with a short introduction into component frameworks. Then, it explains two practically important component frameworks: the Component Object Model of Microsoft, *COM* for short, and the Enterprise JavaBeans framework of Sun Microsystems, *EJB* for short.

Kapitel 6

Description Techniques for Architectures

The central result of the design phase for a software system is its architecture. It would be of great value if we had appropriate notations for capturing and expressing this result in a precise way. The goal is to describe software architectures

- on the right level of abstraction,
- in a standardized way to ease communication between the users of the architectural description,
- based on techniques and tools that allow for consistency checking, visualization, and code generation.

In all of these aspects, architectural descriptions can be compared to programs. Modern programming languages should also support abstract concepts[1] and abstraction mechanisms. However, programming languages must allow for the description of efficient implementations. Thus, the focus of programming languages is more on effectiveness and efficiency. Furthermore, current programming languages mainly provide constructs for the description of components whereas architectural descriptions have to concentrate on the connections between components and their interaction. The components themselves need only be treated by their abstract interface properties, details of their implementation are beyond the scope of architectural descriptions.

Architectural descriptions are essentially used for the communication between people, e.g. between system designers and programmers, whereas programs are mainly used for the communication between man and machine. Thus, architectural descriptions need not be complete and detailed to the last

[1] Ordinary programmers often have only a vague idea of how these concepts are mapped to the computer.

bit. Too much formality can even be a disadvantage by hampering the view to the central properties. On the other hand, an architectural description can as well gain from precision and formality, e.g. to avoid misunderstandings and to allow tool support.

This chapter and the companion article explain three different approaches to describe architectural structures and software architectures:
The *first* approach is based on a diagrammatic, popular language called UML. It provides standardized notations for architectural structures, in particular for those occurring in object-oriented design.
The *second* approach is based on a simple architectural framework and shows how programming languages can be used to describe architectural issues.
The *third* approach is based on the concepts of modern architecture description languages, ADLs for short. An *ADL* is a formal language that is designed for the specification and analysis of software architectures.

Kapitel 7

Designing Software Architectures

It is one thing to know how to describe architectural structures and aspects. Another thing is to design a good system architecture satisfying given requirements. In this chapter, we investigate how software architecture is related to the software development process. Classically, the software development process distinguishes four phases: requirement capture & analysis, design, implementation, and test. The architecture of a software system is mainly worked out in the design phase.

This chapter consists of two parts. First, it relates software design and software architecture. The central question is:

What is the role of architectural structures w.r.t. different design methods and techniques?

In the second part, the chapter considers criteria for the evaluation of architectures. Based on a representative example, it discusses the influence of non-functional and technical requirements on architecture design. The question here is:

What makes a good architecture?

Kurs 1798

Aufgaben zur Kurseinheit 1

Aufgabe 1: Eigenschaften von Softwaresystemen P: 8 + 8 + 8 + 8 = 32

In Abschnitt 2.1.1 werden die Eigenschaften installierter oder im Betrieb befindlicher Softwaresysteme genannt. Ihre Aufgabe ist es, die in Abschnitt 1.1.1 als Beispiele aufgeführten Softwaresysteme auf diese Eigenschaften hin zu untersuchen.
Bitte wählen Sie vier der acht gegebenen Softwaresysteme aus und beschreiben Sie für diese kurz und stichpunktartig, was ihre Identität, Lebenszyklus, Zustand und Verhalten sowie ihre Plattform und Systemumgebung sind.

Aufgabe 2: Zugrunde liegende Strukturen P: 5 x 6 = 30

Abschnitt 1.2.1 nennt eine Reihe von Strukturen zur Beschreibung von Software-Architekturen. Bitte denken Sie einmal über die Sie umgebende Software (nicht nur in Ihrem PC!) nach und betrachten Sie sie mit den Augen des Software-Architekten!
Suchen Sie nach Beispielsystemen, mit denen Sie die unterschiedlichen Strukturen illustrieren können.
Nennen Sie für jede Struktur in der angegebenen Liste:

- ein System mit kurzer Beschreibung
- die wesentlichen Komponenten des Systems bzgl. der Struktur
- die Beziehung der Komponenten zueinander

Dabei können Sie anhand eines Systems auch mehrere Strukturen illustrieren.

Exemplarisch ist hier eine Lösung für den ersten Punkt vorgegeben. Bitte orientieren Sie sich bei Ihrer Lösung an dieser Vorgabe:

Conceptual Structure

Name des Systems: World Wide Web und Internet-Dienste. Bei diesem System handelt es sich um eine Menge von Diensten, die in einem weltumspannenden Netzwerk auf tausenden von Server-Computern bereitgestellt werden und die Millionen von Nutzern besitzen.

Wesentliche Komponenten des Systems:

- Server: Programme, die auf Anfrage Daten liefern oder weiterleiten (oft sog. daemons). Die Gesamtheit aller Serverprogramme auf allen Internet-Servern bildet einen Dienst (z. B. das World Wide Web).
  Beispiele: HTTP-Server, FTP-Server, SSH-Server.
- Clients: Programme, die diese Dienste nutzen, um Daten von den Servern anzufordern oder an sie zu senden.
  Beispiele: Webbrowser, FTP-Programm, SSH-Client, WAP-Browser oder Webcrawler[1].

[1] Webcrawler werden zur Indizierung von Webseiten verwendet; siehe hierzu auch Aufgabe 4.

Beziehung der Komponenten zueinander:

- Client - Server : Der Client stellt Anfragen an einen Server. Der Server beantwortet diese durch Zusenden oder in-Empfang-nehmen von Daten.
  Beispiel: Der Dienst „eMail“ wird von einem Client (in der Regel ein eMail-Programm) benutzt, um die geschriebenen eMails eines Benutzers zwecks Versand an einen Server (ein eMail-Server-Programm) weiterzuleiten sowie um vom Server empfangene, an den Benutzer adressierte eMails in die lokale Benutzungsumgebung des Benutzers zu übertragen. Ein Server empfängt diese Daten und leitet sie an einen weiteren Server (s. u.) weiter oder speichert sie in einem Postfach, falls der Empfänger der eMail bei ihm registriert ist.
- Server - Server: Dient der Weiterleitung von empfangenen Daten zu einem entfernten Client.
  Beispiel: Ein eMail-Server-Programm leitet von einem Client empfangene eMail an einen anderen Server weiter, falls der Empfänger der eMail nicht bei ihm registriert ist.

Aufgabe 3: Charakteristische Merkmale von Softwaresystemen P: 9

In Abschnitt 2.1.2 werden charakteristische Merkmale von Softwaresystemen erläutert. Bitte untersuchen und klassifizieren Sie das „Call Center Customer Care System“[2] im Hinblick auf diese Charakteristika.

Aufgabe 4: Suchmaschinen im Internet P: 9 + 10 + 10 = 29

In dieser Aufgabe betrachten wir Internet-Suchmaschinen.

a) Architektur von Suchmaschinen

Suchmaschinen erfüllen zwei Aufgaben:

- das automatische Sammeln und Archivieren von Daten über WWW-Seiten
- das interaktive Beantworten von Anfragen

Das Sammeln der Daten geschieht nach folgendem Prinzip: Ausgehend von bestimmten Einstiegsadressen suchen spezielle Prozesse (sog. *Gatherer*) das WWW ab, indem sie Links folgen. Aus dem Volltext der besuchten WWW-Seiten wird in der Regel automatisch eine Menge von Schlagworten extrahiert. Diese werden zusammen mit dem URL der WWW-Seite in eine Datenbank eingetragen. Die interaktiven Benutzeranfragen ermöglichen das Abfragen dieser Datenbank.
Skizzieren Sie eine Architektur für eine Suchmaschine, die mit mehreren Gatherern arbeitet und mehrere zeitgleiche Benutzeranfragen unterstützt!
Erläutern Sie die Architektur und stellen Sie sie graphisch dar!
Orientieren Sie sich dabei an den graphischen Notationen des Kurstextes!

[2] Bzgl. der Unterlagen dazu siehe die Studierhinweise.

b) Architektur von Meta-Suchmaschinen

Da normale Suchmaschinen nur einen Bruchteil der im WWW vorhandenen Seiten abdecken, setzen sich die sog. Meta-Suchmaschinen immer mehr durch. Diese unterhalten keine eigene Datenbasis. Stattdessen wird jede Anfrage an eine Reihe normaler Suchmaschinen weitergeleitet. Die Ergebnisse dieser Sekundäranfragen werden geeignet verknüpft und dem Benutzer einheitlich dargestellt.
Entwerfen Sie eine Architektur für eine Meta-Suchmaschine, die sich auf drei normale Suchmaschinen stützt, und stellen Sie diese Architektur graphisch dar!
Erläutern Sie Ihre Entwurfsentscheidungen!

c) Architektur von Suchmaschinen der nächsten Generation

Suchmaschinen der nächsten Generation (sog. Level-3-Maschinen) sind Meta-Suchmaschinen, die dem Benutzer zusätzliche Dienste bieten, wie z.B.

- die Berücksichtigung der Interessen des Anfragers, um die Treffgenauigkeit zu erhöhen
- das Ausblenden gewisser Treffer, z.B. aus Gründen des Jugendschutzes
- das Anreichern der Ergebnisse mit Zusatzinformationen, z.B. über die Struktur des gefundenen Web-Site.

Dazu setzen diese Maschinen auf gewöhnlichen Meta-Suchmaschinen auf und führen unter Verwendung von weiteren Datenbanken und Suchmaschinen eine Nachbehandlung (z.B. Filterung, Anreicherung, graphische Aufbereitung) der von diesen gelieferten Treffer durch.
Erweitern Sie Ihre Architektur aus Aufgabe (4b) für Level-3-Maschinen!
Wiederum sollen Sie Ihre Architektur erläutern und graphisch darstellen.